Exercices C++ (collection #2)

Exercice 1 : Membres et méthodes statiques

Utiliser la dernière version de la classe **FigGeom**.

A. Comment savoir à chaque instant quel est le nombre total d'instances de figures géométriques pendant l'exécution d'un programme? Rajouter un membre statique pour résoudre le problème.

B. Comment être sûr que personne d'autre ne modifie le compteur ? Faites le nécessaire pour le protéger.

C. On veut attribuer à chaque figure créée pendant l'exécution du programme une étiquette unique de gendre "**Figure 1**", "**Figure 2**" etc. que la méthode **Affiche** pourra afficher. Rajouter les attributs et les méthodes nécessaires pour cela.

Exercice 2 : Objets complexes et gestion de la vie des sous-objets

Utiliser la version de la classe **FigGeom** développée dans l'exercice précédent.

A. On souhaite concevoir une classe complexe **CDessin1** composée de 3 figures géométriques comme attributs. Il s'agit d'un triangle, d'un carré et d'un pentagone dont la couleur sera décidée par l'utilisateur de la classe **CDessin1** à sa construction. Ecrire (seulement si nécessaire) le ou les constructeurs nécessaires, idem pour le destructeur et l'opérateur de copie. Rajouter une méthode d'affichage **Affiche** vers un fichier dont le pointeur **FILE*** arrive en argument et un opérateur << avec un rôle similaire. Tester la création, l'affichage, la copie, la destruction de cette classe.

B. On souhaite concevoir une classe complexe **CDessin2** composée de 3 figures géométriques comme attributs. Ces figures auront un nombre de sommets égal à **n**, **n+1** et **n+2** où **n** est choisi par l'utilisateur à la construction de la classe **CDessin2**. Il pourra choisir aussi une couleur commune pour l'ensemble des figures, comme deuxième argument de construction, avec une valeur par défaut blanche. Ecrire (seulement si nécessaire) le ou les constructeurs nécessaires, idem pour le destructeur et l'opérateur de copie. Rajouter une méthode d'affichage **Affiche** vers un fichier dont le pointeur **FILE*** arrive en argument et un opérateur << avec un rôle similaire. Tester la création, l'affichage, la copie, la destruction de cette classe.

Exercice 3 : Gestion souple d'un nombre variable d'objets de même type

Trouver une architecture de données (objets) permettant de créer et garder ensemble un nombre variable d'objets de même type en les initialisant à leur création dynamique. Cette architecture d'objets doit permettre la destruction d'un objet parmi ceux qui existent, le rajout d'un autre objet (s'il reste des places disponibles) sans que cela perturbe la vie des autres objets.

A. Prenez comme exemple des objets de type **FigGeom** ou **CFract**. Ecrire une fonction qui prend comme argument le nombre d'objets à créer et qui crée des fractions (avec des valeurs 1/1, 1/2, 1/3 etc.) puis les détruit.

B. Une fois le mécanisme compris, modéliser une carte de jeux **CCarte** et une pile de cartes **CPile**. Prévoir des constructeurs pour chaque classe, en empêchant que l'on puisse créer deux carte identiques (ou d'une manière plus générale, seulement la pile peut créer des cartes). Une pile sera initialement soit vide soit remplie d'un jeu complet de 52 cartes (prévoir les versions de construction). Prévoir aussi l'affichage des contenus et la possibilité de passer une carte d'une pile à une autre (il faut garantir que l'utilisateur ne peut ni détruire, ni créer une carte directement).

Exercice 4 : Encapsulation d'API Windows

Le but de l'exercice est de concevoir des chronomètres encapsulés dans des objets C++, en utilisant les API Win32/64 pour la relève du temps courant (inclure <windows.h> pour avoir accès aux API).

A. Concevoir une classe **CChrono** qui mesure le temps entre sa création et le moment où l'utilisateur appelle une des méthodes suivantes (n'importe quand ou combien de fois) : **GetTimeMs**, **GetTimeUs** ou **GetTimeNs**. Chacune des méthodes retourne le laps de temps (en tant que **DWORD**, c'est à dire **unsigned long**) exprimé en millisecondes, microsecondes ou respectivement en nanosecondes. Prévoir une méthode **Restart** qui repositionnera l'origine du laps de temps. Utilisez l'API **GetTickCount** pour avoir le temps courant.

Testez avec cette nouvelle classe la durée d'affichage d'un message à l'écran ou la durée effective d'appel à la fonction **Sleep** avec un argument égal à 20.

B. Comme la finesse du chronomètre encapsulé par **CChrono** n'est pas si bonne, concevez une nouvelle classe **CChronoHR** (comme **H**aute **R**ésolution) qui aura exactement la même interface (ensemble de méthodes et attributs publics) alors que l'implantation sera différente. Elle utilisera les API suivantes qui ont une meilleure précision : **QueryPerformanceCounter** et **QueryPerformanceFrequency**. Regarder l'aide MSDN pour plus de détails.

Ecrire 2 fonctions de test pour un objet soit de type **CChrono** soit de type **CChronoHR**. Elle doit afficher une dizaine de fois (**nb_clk**) des messages à l'écran et de mesurer le laps de temps par rapport au début de la fonction, et de mémoriser ces laps dans un tableau de **DWORD** de taille **nb_clk** (à définir comme constante globale). Prévoir un paramètre **bool** pour cette fonction qui déclenchera (s'il est vrai) dans la boucle l'appel supplémentaire de **Sleep(20)**.

Exercice 5 : Les flux (streams) d'entrée / sortie : Modification de classes

Ne jamais mélanger dans le même programme les flux C (**stdio.h**) avec les flux C++ de STL (**iostream**) !

A. Modifier la version initiale de **FigGeom** pour qu'elle puisse afficher vers la sortie **cout** et vers un fichier C++. Respecter l'alignement vertical des coordonnées à l'affichage. Finalement, exporter cette nouvelle classe dans **FigGeom.h** sans promouvoir (casser) le *namespace* **std** au niveau global.

B. Modifier la version initiale de **CTriPascal** pour qu'elle puisse s'afficher vers un flux **ostream**. Tester avec **cout**.

C. Surcharger l'opérateur << pour afficher un objet de type **CFract** vers un flux de type **ostream**. Quel droit d'accès pour la surcharge ? Peut-on se passer du qualificatif **friend** ? Même chose pour **FigGeom** et **CTriPascal**.

D. Modifier la classe **CPoly** pour faire afficher le polynôme vers un **ostream** à l'aide de la méthode **Show** et l'opérateur <<. Modifier aussi les lignes de tests fournis précédemment.

Exercice 6 : Flux et membres statiques

On veut disposer d'un fichier de log automatique **"cfract.log"** qui pourra garder l'historique de la création et de la destruction de tout objet de type **CFract**. La solution "objet" pour réaliser cela passe par la déclaration comme attribut statique d'un pointeur vers un **FILE** (si l'on veut utiliser un flux C) ou d'un **ofstream** (si l'on veut utiliser un flux de sortie C++). Pour cet exercice on va préférer la deuxième solution. Cet attribut sera unique (le même) et accessible pour tous les objets de la classe (empêcher l'accès au *stream* de l'extérieur).

Utilisez-le dans chaque constructeur et destructeur en affichant l'endroit, le contenu de l'objet et son adresse. Tester en créant plusieurs objets **CFract**, en appelant aussi des fonctions de type suivant avec des arguments du type demandé (à vous de remplir les corps des fonctions avec un code très court mais correct):

```
void Fonct1(CFract f) {}
void Fonct2(CFract& rf) {}
void Fonct3(CFract* pf) {}
CFract Fonct4(CFract rf) {}
CFract Fonct5(CFract& rf) {}
CFract& Fonct6(CFract* rf) {}
CFract& Fonct7(CFract& rf) {}
```

Exercice 7 : Modélisation d'un match de tennis [Objets complexes]

On veut modéliser le calcul du score d'un match de tennis en fonction de la succession des balles (coups, points) gagnées par l'un et l'autre des deux joueurs. Comme sortie on utilise les flux C++ (**iostream**). Le calcul du score se décompose en plusieurs composants (version simplifiée des vraies règles !) :

le jeu : gagné par le joueur qui a plus de 3 balles et au moins 2 balles de plus que son adversaire;

le set : gagné par le joueur qui a plus de 5 jeux et au moins 2 jeux de plus que son adversaire;

le match : il est gagné par le joueur qui a gagné en premier 3 sets;

Le score affiché après chaque balle comprend le nombre de sets gagnés par chaque joueur dans le match, le nombre de jeux gagnés par chaque joueur dans le set en cours et le score dans le jeu courant. Ce dernier, pour des raisons historiques, s'exprime par la succession 0, 15, 30, 40. Si les deux joueurs ont plus de 4 balles mais moins de deux balles de différence on affiche "Avantage" pour le joueur qui a plus de balles gagnées ou "Egalité" en cas d'égalité.

A. Concevoir une classe **CJeu** qui puisse permettre de mémoriser, calculer et afficher le score d'un jeu avec l'interface suivante :

- Constructeur sans paramètres.

- Méthode **NewPoint** appelée à la fin de chaque balle avec un paramètre booléen vrai si c'est le joueur **J1** qui a gagné le coup (la balle). Elle doit retourner faux tant que le jeu n'est pas fini et vrai dès qu'un joueur a gagné le jeu. Après la fin du jeu tout autre appel de cette méthode ne modifiera plus l'état de l'objet (on retourne aussitôt avec une valeur vraie).
- Méthodes **WinJ1** et **WinJ2**, sans paramètres, qui retournent un booléen vrai quand le joueur **J1** ou respectivement **J2** ont gagné le jeu.
- Méthode **Score** qui affiche vers un objet de type **ostream** (dont la référence arrive en paramètre) le score actuel du jeu (version historique).
- Constructeur de copie, opérateur d'attribution et destructeur seulement si nécessaire.

Conseil pour implémentation : prévoir comme attributs deux booléens **win_j1** et **win_j2** qui indiqueront qui a gagné ainsi que deux entiers **nb1** et **nb2** qui indiqueront les points gagnés par chaque joueur.

B. Concevoir les classes **CSet** et **CMatch** avec la même interface et presque les mêmes attributs pour mémoriser, calculer et afficher le score d'un set et respectivement d'un match. Conseil pour l'implémentation: **CSet** aura **MAX_JEUX** sous-objets de type **CJeu** et **CMatch** aura **MAX_SETS** sous-objets de type **CSet**.

C. Tester individuellement les classes conçues puis "jouer" un match où le joueur J1 à 60% de chance de gagner une balle contre le joueur J2 (utiliser la fonction C **rand**). Afficher le score à la console après chaque balle jouée.

D. Surcharger l'opérateur de décalage à gauche "<<" de telle façon qu'il puisse déclencher un affichage du score d'un objet de type **CMatch** vers un objet de type **ostream**. Modifier la séquence de test pour utiliser le nouvel opérateur à la place de la méthode **Score**.

Exercice 8 : Modélisation d'un jeu de Domino [Objets et conteneurs sans ou avec gestion de ressources]

A. On souhaite modéliser des pièces de domino par une classe **CDomino**. Mémoriser dans deux attributs appelés **l** (comme left) et **r** (comme right) le nombre de points de la partie gauche et droite de la pièce. Utiliser le plus petit type natif qui peut contenir des nombres entre 0 et 6. Les seuls moyens de création d'un objet domino seront :

- soit par un constructeur qui apporte comme paramètres ces points,
- soit par un constructeur de copie.

Ecrire, seulement si nécessaire, le constructeur de copie, l'opérateur d'attribution et le destructeur.

Prévoir une méthode **Affiche** qui affiche le contenu d'un domino vers le flux C++ sortant dont la référence arrive en argument, ainsi qu'un opérateur "<<" d'insertion vers un flux sortant. L'affichage se fera sous le format suivant : [n_l:n_r] où n_l est la valeur de la partie gauche et n_r celle de la partie droite.

Prévoir un opérateur "**==**" qui retourne un booléen représentant le résultat de la comparaison entre l'objet courant et un autre.

Prévoir une méthode **Ranger** qui s'assure que la valeur de gauche est inférieure ou égale à la valeur de droite, en les permutant si nécessaire.

Prévoir un opérateur "~" qui renvoie (sans modifier l'objet courant) une copie de la pièce retournée (on permute les parties gauche - droite).

Prévoir une méthode **Coller** et un opérateur "^" qui retournent vraie seulement si la partie droite de l'objet courant est égale à la partie gauche de l'objet qui arrive en argument.

Tester l'ensemble de la classe.

B. Concevoir une classe **CPile** qui modélise une pile de dominos (entre 0 et le maximum de pièces) : elle aura un tableau dynamique de **CDomino** (**pile**) dont la taille sera mémorisée dans un autre attribut (**sz_pile**). Attention à la gestion de mémoire !

Faire en sorte que l'on puisse construire des objets de type **CPile** à partir de rien (pile vide), d'un objet **CDomino** ou d'un autre objet de type **CPile**. Ecrire, si nécessaire, le constructeur de copie, l'opérateur d'attribution et le destructeur.

Prévoir l'accès en lecture seule à la taille de la pile ainsi qu'aux pièces.

Prévoir une méthode **Affiche** et un opérateur "**<<**" qui afficheront toutes les pièces de la pile dans le flux C++ sortant dont la référence arrive comme paramètre.

Concevoir un opérateur "**+=**" qui rajoute à l'objet courant le contenu d'un autre objet de type **CPile**.

On souhaite rajouter aussi un seul objet **CDomino** à un objet **CPile** à l'aide de l'opérateur +=. Que faut-il faire ?

Concevoir un opérateur "+" qui "additionne" 2 objets de type **CPile** en produisant un nouvel objet **CPile** qui contient les 2 pièces opérandes.

Concevoir une méthode **Melanger** qui mélange d'une manière aléatoire (en utilisant **rand**) les pièces de dominos de la pile.

Concevoir une méthode **Extraire** à deux paramètres entiers jouant le rôle d'indices de début et de fin. Elle extrait les pièces situées entre ces indices (inclusivement) et retourne un nouvel objet de type **CPile** avec les pièces en question. Toute erreur d'indice retourne un objet vide. A la fin, l'objet courant ne possède plus les pièces extraites.

Concevoir une méthode **Extraire** à zéro paramètres qui retourne un nouvel objet **CPile** contenant la première carte de la pile (ou rien si l'objet courant est vide).

Tester l'ensemble de la classe.

C. Dériver la classe **CPile** vers une classe **CJeuDomino** qui possède un constructeur sans paramètres qui "fabrique" et garde dans l'ordre les 28 pièces de domino (de [0:0] à [6:6]).

Tester cette classe en respectant le scénario suivant :

- créer un objet **CJeuDomino** avec toutes les pièces,
- afficher son contenu à la console,
- mélanger son contenu, puis réafficher son contenu à la console,
- extraire du jeu quatre piles de 7 pièces chacune,
- afficher le contenu des 5 objets dans un fichier "piles.txt".

Rajouter une méthode **Chaine** qui mélange d'abord les pièces puis commence à former une chaîne en commençant par la première pièce. Elle cherche parmi les pièces suivantes celle qui "colle" à la première, la ramène en deuxième place puis elle continue à rechercher une autre pièce à coller parmi celles qui restent etc. Chaque pièce collée est affichée à la console. Le processus s'arrête quand on ne trouve plus de pièces à coller ou quand il n'en reste plus.